Chapter 3. Types and Expressions

The last chapter introduced programming with some rather simple examples. The main programming elements that we used were methods provided in Processing and our programs consisted of calls to these methods to perform certain actions such as opening a display window of a certain size, drawing various geometric shapes such as points, lines, rectangles, and ellipses in this window along with images and fonts. In this chapter we focus on data, how it can be stored and processed in a program.

3.1. Example: Two-Dimensional Design and Layout

Photographers compose photographs with careful attention to fundamental design principles. Creating such works of art may seem to be effortless, even arbitrary, but this is far from the case. Consider the photograph shown in Figure 3-1. It includes a sailboat and the sun setting on the horizon. Rarely do photographers place the item or items of interest directly in the center of the photo or in arbitrary locations. Rather, they are positioned in certain pleasing proportional relations with one another.

Figure 3-1. A sample photograph

$$\frac{a+b}{a} = \frac{a}{b} = \varphi$$

Figure 3-2. The golden ratio

The question of what constitutes a "pleasing" proportional relationship has fascinated philosophers, mathematicians, artists and others for centuries. One of the most famous answers to this question dates back to the 4th century BC when the Ancient Greeks studied and used what has come to be known as the *golden ratio*, *golden mean*, *golden section*, or *sectio divina* ("divine section") because it occurs in so many places in nature, architecture, and art, as well as in other phenomena. This ratio is defined by the problem of dividing a line segment so that the ratio of its length to the larger part is equal to the ratio of the larger part to the smaller part, as illustrated in Figure 3-2 .

This golden ration is commonly denoted by a Greek lower case phi, written **φ.** If we substitute φ for the ratio *a*/*b* in the first part of the formula in Figure 3-2, we obtain

$$1 + \frac{1}{\varphi} = \varphi$$

which can be rewritten $\varphi^2 - \varphi - 1 = 0$. Solving this using the quadratic formula from algebra gives the following irrational number as the unique positive solution of this equation:

$$\varphi = \frac{1+\sqrt{5}}{2} = 1.6180339887\ldots$$

This golden ratio crops up in an incredible number of places and, in particular, has been used as a design tool in art and architecture.

In this chapter we will study the basic data types provided in Processing and the kinds of expressions that can be used. We will use them to build a simple tool for analyzing proportional design — annotating an image by drawing lines overlaying the image that indicate pleasing proportional relationships and focus points based on the golden ratio.

3.2. Types

In nearly all programming languages, each data value must have a specified **data type** that informs the computer how that value is to be represented, stored, and manipulated. To allow Processing to ensure that it is handling all data items properly, it requires that the program specify the type of each data item before it is used. This is done by using a **declaration** of that item.

Processing provides two kinds of types: **primitive types** and **reference types**. Primitive types are the types used to specify the basic data elements in Processing and in most programming languages: integers, real numbers, boolean (or logical) values, and single characters. Table 3.1 lists Processing's primitive types, giving their names, brief descriptions, ranges of values they may have, and the number of bits required to store such values.

Table 3-1 Processing's Primitive Types

Name	Description	Values	Number of Bits
`byte`	Very small integers	-128 (=(2^7) through $+127$ (= $2^7 - 1$)	8
`short`	Small integers	-32768 (= -2^{15}) through $+32767$ (= $2^{15} - 1$)	16
`int`	Integers	-2147483648 (= -2^{31})through $+2147483647$ (= $2^{31} - 1$)	32
`long`	Large integers	-2^{63} through $+2^{63} - 1$	64
`float`	Real numbers	$-3.4028235 \times 10^{38}$ through $+3.4028235 \times 10^{38}$ (approx.)	32
`double`	Large real numbers	$-1.7976931348623157 \times 10^{308}$ through $+1.7976931348623157 \times 10^{308}$(approx.)	64
`char`	Single characters	Letters, numerals and other characters with Unicode representations as integers from 0 through 65535 (= $2^{16} - 1$)	16
`boolean`	Logical values	true or false	1

Reference types are built from other types. Most often, these other types are **classes**, discussed in Chapter 7; instances of classes are called **objects**. *The type of an object must be a reference type*. Reference types include `String` for text strings, and `PImage` for images. These are but two of hundreds of reference

types that Processing provides. (Descriptions of many of the most important types can be found in the Processing web reference library. [1])

The set of primitive types is fixed and Processing reserves their names (see Table 3.2) *as* ***keywords*** *that may not be used for other reasons in a program*. The collection of reference types, on the other hand, can be expanded by adding programmer-defined types and Processing does not reserve their names as keywords. One consequence of this is that reference types must be explained to the compiler, either in the program file itself or in a predefined library.

3.2.1. Literals

A data value of a given type, coded directly into a program, is called a **literal**. Table 3.2 gives some examples of literals and their types.

Table 3-2 Examples of Literal Values

Type	Examples
`int`	`-1, 0, 1, 42, 1024`
`float`	`-2.5, 0.0, 0.001, 3.0e8, 1.0E-4`
`boolean`	`true, false`
`char`	`'A', 'a', ' ', '.', '\n'`
`String`	`"hello", "", "You are here."`

Processing is able to determine the type of a literal value from its form, more commonly called its *syntax*:

- Numeric literals without decimal points are `int` (or `byte` or `short` or `long`) values.
- Numeric literals with a fixed decimal point (denoted as *m.n* with integer part *m* and decimal part *n*, e.g., `123.4`) or scientific notation (denoted *x*E*n* or *x*e*n* with an integer or fixed point *x* and an integer exponent *n*, e.g., `1.234e2` denotes 1.234 * 10^2 or 123.4) are `float` values.
- The keywords `true` and `false` are `boolean` values.
- Individual characters enclosed by single quotes are `char` values.
- Sequences of characters enclosed in double quotes are `String` literals.

By default, positive and negative integers are treated as literals of type `int`. Appending the letter `L` or `l` to a literal (e.g., `-30L`, `0L`, `+365L`) causes it to be treated as `long` instead of `int`. Base-8 and base-16 representations of numbers are also allowed: a sequence of digits that begins with `0` is interpreted as an *octal* (i.e., base-8) integer, provided that the digits are octal digits 0, 1, . . . 7; a sequence of digits preceded by `0x` is interpreted as a *hexadecimal* (i.e., base-16) integer with the usual letters `A`, `B`, . . . , `F` (or lowercase equivalents) used for ten, eleven, . . . , fifteen, respectively. The following examples illustrate this:

[1] http://processing.org/reference/

`12` has the decimal value $12_{10} = 1 \times 10^1 + 2 \times 10^0$

`012` has the octal value $12_8 = 1 \times 8^1 + 2 \times 8^0 = 10$

`0x12` has the hexadecimal value $12_{16} = 1 \times 16^1 + 2 \times 16^0 = 18$

By default, floating point numbers written using either fixed decimal or scientific notation are treated as literals of type `float`. Appending the letter D or d to a literal (e.g., `123.4d`) causes it to be treated as a `double` instead of a `float`.[2]

3.2.2. Identifiers

In the previous chapter we used literal values in all of the example programs. While this approach is relatively easy to understand, it is not very powerful because it limits the programs' flexibility. The power of programming comes from its ability to model data and manipulate and process it in various ways. This requires that the program use names for its data values so that it can refer to them, perhaps modify them, and process them throughout its execution. These names are called **identifiers**.

For example, Processing programs usually specify the dimensions of their display window and in the previous chapter we used literal integer values for this purpose — for example,

```
size(300, 225);
```

This statement uses literal values to set the width and height of the display window to 300 and 225 pixels, respectively. Whenever the program needs to refer to the width of the image, it repeats the literal `300`:

```
point( random(300), random(225) );
```

This statement correctly draws a point at some random location in the display window. But if we now change the dimensions of the display window, say with

```
size(150, 100);
```

then the values in the call to the `point()` method must also be changed accordingly. We would have to manually modify our program wherever these values for the display window's width and height are used.

Doing such re-programming is not a good practice because it can easily lead to programming errors. For example, we must change all the occurrences of 300 and 225 consistently; missing or mistyping even one would result in an incorrect program. Also, some places where one of these literals is used might not actually refer to one of the display window's dimensions at all but instead is used to specify a high color intensity value that should not be changed. Doing a global replacement of 300 with 150 and 225 with 100 would then not be appropriate and would result in an incorrect program.

The problems related to *inconsistent* or *inappropriate* changes can be addressed by using *named* data values to represent the width and the height of the display window and then use these names instead of the literals throughout the program:

---

[2] Java, the language on which Processing is based, defaults to using `double` values. When a `float` value is desired in Java, an `f` designator can be used, e.g., `123.4f`.

```
// width and height are declared and assigned values here
size(width, height);
// some intervening code
point( random(width), random(height) );
```

This code uses the identifiers `width` and `height` to refer to values representing the width and height of the display window. Any time the program needs to use these values, it uses the appropriate variable.

One additional advantage of using named variables is *improved readability*. For example, the method call `random(300)` assumes that programmers know what the 300 represents, while the call `random(width)` makes the nature of the representation much clearer.

In Processing, *an identifier must begin with a letter or underscore (_), which may be followed by any number of additional letters, digits or underscores. This gives programmers the flexibility to use meaningful names that suggest what they represent.* Although different programmers have different naming styles, there are certain **naming conventions** that are commonly practiced:

- **Variables** – *Names for storage locations for values that can change are given in lower case. If the name consists of multiple words, the first letter in each word following the first word is capitalized*, e.g., `width`, `currentHeight`, `outputWindowHeight`. (This is sometimes referred to as "camelback" notation.)
- **Constants** – *Names for storage locations for values that cannot change are given in all uppercase. If the name consists of multiple words, the words are separated by an underscore*, e.g., `PI`, `TWO_PI`, `MAX_OUTPUT_WIDTH`.
- **Methods** – *Names for methods are like variable names but are followed by parentheses*, e.g., `size()`, `strokeWeight()`,
- **Classes** – *Names for classes are like variable names but are capitalized*, e.g., `String`.

These naming conventions are not enforced by Processing but using them is recommended because they make it easy to determine at a glance whether a name refers to a variable, constant, method or class.

3.2.3. Declaration Statements

Keywords in Processing have predefined meanings but other identifiers in a program do not. For all such identifiers a program must provide information to the compiler about each one before it is used. This is done by means of **declaration statement**s.

Primitive Types. Most programs store data values in memory locations from which these values can be retrieved and processed. Locations whose values may change are called *variables* and are declared using a **variable declaration statement**. To illustrate how this is done, consider the following code:

```
int width = 300;
int height = 225;
size(width, height);
// some intervening code ...
point(random(width), random(height));
```

This code segment **declares** two variables of type `int` named `width` and `height`, and **initializes** their values to 300 and 225, respectively. Processing represents these two variables as two named memory locations.

```
width  [300]
height [225]
```

This allows the programmer to set the width value to 300 in one place and refer to it by name many times later in the program. The program can change the value but the name remains the same. This way, if the programmer modifies the declaration of `width` to change the width of the output window, the value at the memory location named `width` will change. All further references to that variable get the new value thus avoiding the dangers of inconsistency and incorrect modifications.

The general form of a variable declaration is as follows:

Variable Declaration

type variableName;

or

type variableName = expression;

or

type identifierExpressionList;

where

- *type* is the data type that the variable should represent;
- *variableName* is the identifier used to refer to the variable's value;
- *expression* is any expression that returns a value whose type is *type*.
- *identifierExpressionList* is a comma-separated list of one or more identifier expressions of the form *variableName* or *variableName = expression*.

In the first form, the value used to initialize *variableName* will be 0 for primitive numeric types, `false` for boolean types. In the second form, the value of *expression* will be evaluated and used to initialize *variableName*. This initialization is optional for variable declarations. The third form means that we can combine several declarations of the same type into a single declaration; for example, the declarations of the variables `width` and `height` in the preceding code segment could be combined into a single declaration:

```
int width = 300, height = 225;
```

Some programs may need data values that should not change. Such values are represented as **constants**. Processing provides many predefined constants such as `PI` and `TWO_PI` which represent the mathematical constants π and 2π, respectively. Processing allows programs to access the value of a constant just as with the value of a variable but it prevents them from ever changing its value. Though

Processing does not provide a constant representing the irrational number φ, the program can specify an approximate value.

```
final float PHI = 1.6180339887;
```

The general form of a constant declaration is as follows:

Constant Declaration

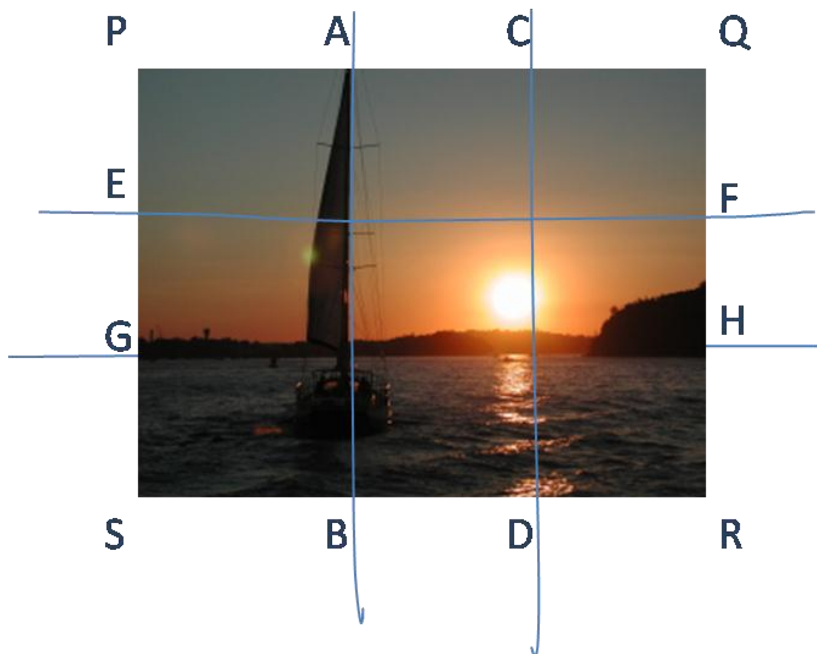
```
final type identifier = expression;
```

- `type` is the data type that the identifier should represent;
- `identifier` is used to refer to the variable's value;
- `expression` is any expression that returns a value whose type is `type`. Note here that the initialization part of the constant declaration is required.

It is generally considered good practice to place all constant declaration statements at the beginning of the program.[3] This makes it easy to locate them should their values need to be reprogrammed.

3.2.4. Example Revisited

Using variables and constants, we can design a program that annotates the photograph shown in Figure 3-1 using divisions based inon the golden ratio. We would like our annotations to look like the lines shown in the diagram shown to the right.



Here, the x and y coordinate values for the annotations are computed using the following formulae.

$$PA = CQ = width / \varphi$$

$$PE = GS = height / \varphi$$

With these values in mind, we can specify the desired behavior using the following algorithm.

Given:

- The constant `PHI` represents the golden ratio.
- We have loaded a bitmap image in the `data` sub-directory.
- `WIDTH` and `HEIGHT` hard-code the dimensions of the image to 300 x 225 pixels respectively.

[3] At the beginning of the block in which they are used, for more complex programs as in the next chapter.

Algorithm:

1. Create a display window that is `WIDTH` x `HEIGHT` pixels.
2. Load and display the image.
3. Draw a vertical line (AB) with the x coordinate = `WIDTH / PHI`
4. Draw a vertical line (CD) with the x coordinate = `WIDTH – (WIDTH / PHI)`.
5. Draw a horizontal line (EF) with the y coordinate = `HEIGHT / PHI`.
6. Draw a horizontal line (GH) with the y coordinate = `HEIGHT – (HEIGHT / PHI)`.

This algorithm is similar to the ones specified in Chapter 2, but has added a set of "given" or assumed conditions. Algorithms make assumptions that are critical to their operation, and these assumptions should be specified clearly as shown here. For example, this algorithm, and the program implemented from it, assumes that the width and height of the image are known in advance and hard-coded into the program. If this assumption is violated, for example if we try to load an image that isn't 300x225 pixels, then the program will not display the correct annotations. It is important for anyone hoping to use this algorithm that they can, as it were, use it for any image they'd like so as long as the image is 300x225 pixels. The algorithm can be implemented as shown below along with its rendered output.

```
final float PHI = 1.6180339887; // approximation of the golden ratio
final int WIDTH = 300, HEIGHT = 225;  // the image's width, height

size(WIDTH, HEIGHT);
image(loadImage("sydneyHarbor-300x225.jpg"), 0, 0);

stroke(125);
float aWidth = WIDTH/PHI, aHeight = HEIGHT/PHI;
line(aWidth, 0, aWidth, HEIGHT);                      // line AB
line(WIDTH - aWidth, 0, WIDTH - aWidth, HEIGHT);      // line CD
line(0, aHeight, WIDTH, aHeight);                     // line EF
line(0, HEIGHT - aHeight, WIDTH, HEIGHT - aHeight);   // line GH
```

Note that the line of the sailboat's mast roughly matches the vertical line joining A and B and the line of the horizon roughly matches the horizontal line joining G and H.

This program illustrates another advantage of using named variables. The variables `aWidth` and `aHeight` represent the larger value of the golden ratio in the horizontal and vertical dimensions respectively. Each of these values is computed once, using a variable declaration statement (numeric expressions are discussed in the next section) and then used multiple times as arguments to the line drawing method. This way the program performs the division operation only once, and then reuses the value multiple times, thus reducing the amount of computation required to execute the program.

3.2.5. Using Reference Types

Although the primitive types discussed so far are adequate to represent simple values like numbers and characters, they are not adequate to represent more complex objects like audio clips, bitmap images and bouncing balls. To represent such objects, Processing allows programmers to create new types, which is done by creating classes. Types created from classes are called *reference types*.

We discuss creating new classes in some detail in a later chapter. Here, it is important to understand how to use reference types created by other programmers. Pre-defined reference types that we will find useful in this text include bitmap images and audio clips.

Reference types differ from primitive types in several ways. One difference concerns how the type definitions are loaded. Primitive types are built into the language, so they are readily available in a program and we don't have to load them explicitly. Reference types, on the other hand, are generally not built-in and must be imported into a program. For example, to use the `AudioSnippet` class, which represents a short audio clip, the programmer must import Minim's audio snippet class definition, as shown here:

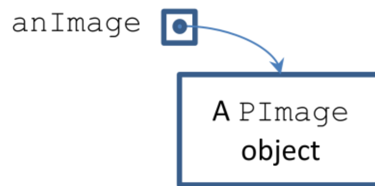
```
import ddf.minim.AudioSnippet;
```

This import statement tells Processing to find and load the class definition found in `ddf.minim.AudioSnippet`. As with constant declarations, import statements should appear external to any method and at the top of the program. With this definition, the user can create and play audio clips as shown below.

Another difference between reference types and primitive types is in how values are created for each type. Because primitive types use literals for their values and the meanings of literals are built into the Processing compiler, primitive type values are predefined. By contrast, there are no pre-defined, literal values for most reference types so they must be created. Creating reference objects is generally done using the `new` operation, which we will discuss in detail in a later chapter, but it can also be done using a specially-defined loading method for the given type. For example, in the previous chapter we used the `loadImage()` method to create a new reference object of type `PImage`. This allows us to declare, initialize and display a bitmap image as shown in the following code:

```
PImage anImage;
anImage = loadImage("anImageFilename.jpg");
image(anImage, 0, 0);
```

Here, the loadImage() method creates a reference object of type `PImage`, which is assigned as the value of the variable `anImage`. No import statement is required for `PImage` because Processing automatically imports the `PImage` class definition. We might picture the storage for this object as follows:



Here, the variable `anImage` contains a reference rather than a primitive value.

We will consider these aspects of reference types in more detail in later chapters, but there is one other difference between reference and primitive types that we want to use now. Once it is created, an *object* — a data value whose type is a reference type — has properties that a primitive type value does not. Objects can provide both data attributes and methods and these properties can be accessed using *dot notation*. The general form of the dot notation is as follows:

Dot Notation

```
objectName.methodName(argumentList)
objectName.attributeName
```

- *`objectName`* is the name of the object that is responding to the message;
- *`methodName`* is the name of the method being called;
- *`attributeName`* is the data attribute being referenced;
- *`argumentList`* is the argument list required by the method.

Reference objects of type `PImage` provide a number of useful properties. For example, we can access the width and height of the bitmap image through its `width` and `height` attributes:

```
anImage.width
anImage.height
```

We can also access the pixel value at coordinates (15, 10) using the `get()` method:

```
anImage.get(15, 10);
```

The PImage class provides other useful properties as well. See the Processing reference manual entry for a complete specification.[4]

[4] http://www.processing.org/reference/PImage.html

3.2.6. Example Revisited

One weakness of the previous version of the image analysis program is that the dimensions of the image are *hard-coded* into the program. The program only works for images whose dimensions are 300 × 225. We can improve the program by explicitly using the `PImage` reference type. For example, the image of the fish-market pelican shown below is larger than the sailboat image in Figure 3-1 but we'd like to annotate it in a similar manner. We can accomplish this by, instead of hard-coding the image dimensions, using variables to store these dimensions and use these variables as arguments to the `size()` method; this allows the program to load and annotate images of any size. The following algorithm is an improved version of the one used in Section 3.2.4.

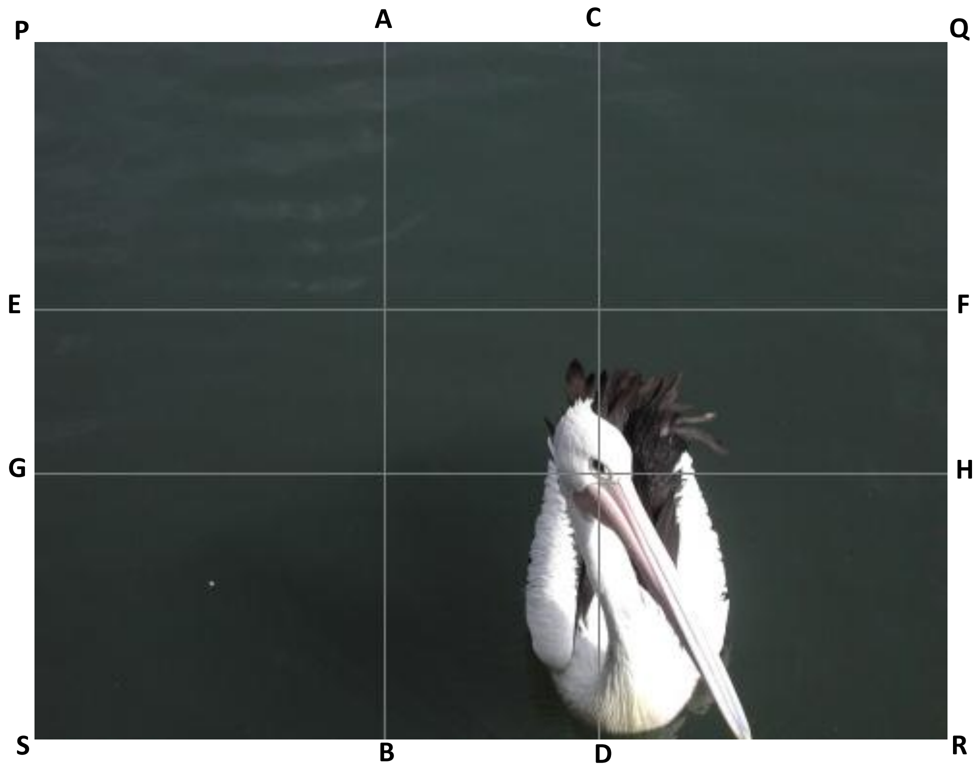
Given:

- The constant `PHI` represents the golden ratio.
- We have loaded a bitmap image in the `data` sub-directory.

Algorithm:

1. Load the image.
2. Set `width` = the image's width and `height` = the image's height.
3. Display the image.
4. Create an display window that is `width` x `height` pixels.
5. Draw a vertical line with the x coordinate = `width / PHI`.
6. Draw a vertical line with the x coordinate = `width - (width / PHI)`.
7. Draw a horizontal line with the y coordinate = `height / PHI`.
8. Draw a horizontal line with the y coordinate = `height - (height / PHI)`.

This algorithm is similar to the one shown in Section 3.2.4 except that it separates the loading from the display of the image. This allows it to use the dimensions of the image to size the display window before displaying the image, which gives the program more flexibility. The algorithm can be implemented as shown here.




```
final float PHI = 1.6180339887; // approximation of the golden ratio

PImage theImage = loadImage("fishMarketPelican-500x375.jpg");
int width = theImage.width, height = theImage.height;

size(width, height);
image(theImage, 0, 0);

stroke(125);
float aWidth = width/PHI, aHeight = height/PHI;
line(aWidth, 0, aWidth, height);                     // line AB
line(width - aWidth, 0, width - aWidth, height);     // line CD
line(0, aHeight, width, aHeight);                    // line EF
line(0, height - aHeight, width, height - aHeight); // line GH
```

This program constructs a `PImage` object using `loadImage()`, stores it in a variable `theImage` and uses the dot notation to retrieve the image's width and height from `image`. It stores the width and height values in the `width` and `height` variables and uses these variables as arguments to the `size()` method, thus initializing the sketch size appropriately for each image. Note that the pelican's eye is placed at the intersection lines CD and GH.[5]

[5] Using variables as arguments to the `size()` method prevents Processing's export routine from determining the intended dimensions of the sketch. To properly support the export of applets or applications, the arguments to `size()` must be literal values. This text generally ignores this restriction in order to support examples such as this one and to present a more standard view of the use variables.

For those interested in audio processing, we could use the `Minim` reference type to play background music while the image is being displayed. Starting with the code shown above, we make two changes. First, at the very top of our program we add import statements to tell Processing to load the Minim reference type:

```
import ddf.minim.Minim;
import ddf.minim.AudioSnippet;
```

Second, we add the following code to play an MP3 file containing Barber's "Adagio for Strings" in the background.

```
Minim minim = new Minim(this);
AudioSnippet snip = minim.loadSnippet("AdagioForStrings.mp3");
snip.play();
//...
```

As is the case with image files, the audio file must be stored in sketch's `data` sub-directory.

3.3. Expressions

A primitive expression is a sequence of one or more primitive-type objects called operands, and zero or more operators that combine to produce a value. Thus `12` is a primitive expression consisting of one operand (`12`) and no operators, producing the `int` value twelve. Similarly, `2.2 + 3.3` is a primitive expression with two operands (`2.2` and `3.3`), one operator (`+`), and produces the float value `5.5`. The type of the value produced by an expression is called the type of the expression. Expressions that produce an `int` value are called `int` expressions, expressions that produce a `float` value are called `float` expressions, and so forth.

3.3.1. Numeric Expressions

In Processing, addition and subtraction are denoted by the usual `+` and `-` signs. Multiplication is denoted by `*`, which must be used for every multiplication. That is, to multiply `n` by `2`, we can write `2*n` or `n*2` but not `2n`. Division is denoted by `/`, which is used for both real and integer division. Another operation closely related to integer division is the **modulus** or **remainder** operation, denoted by `%`, which gives the remainder in an integer division. The following table summarizes these operators.

Table 3-3 Numeric operators

Operator	Operation
+	addition, unary plus
-	subtraction, unary minus
*	multiplication
/	real and integer division
%	modulus (remainder in integer division)

For the operators +, -, *, and /, the operands may be of any primitive integer or real type. If both are integer, the result is integer, but if either is real, the result is real. For example,

```
2 + 3 = 5                 2 + 3.0 = 5.0

2.0 + 3 = 5.0             2.0 + 3.0 = 5.0

7.0 / 2.0 = 3.5           7 / 2 = 3
```

It is important to understand the difference between integer and real division. In the expression `3 / 4`, both operands (`3` and `4`) are integers, so integer division is performed producing the integer quotient `0`. By contrast, in the similar expression `3.0 / 4`, a real operand (`3.0`) is present, so real division is performed producing the real result `0.75`. One of the common problems for beginning programmers is to *remember that the value of* `1/n` *is* 0 *if* `n` *is an integer different from –1, 0, or 1.*

Integer division produces both a quotient and a remainder and Processing uses one operator (`/`) to give the integer quotient and another operator (`%`) to give the remainder from an integer division. The following are some examples:

```
9 / 3 = 3                 9 %  3 =  0

86 / 10 = 8               86 % 10 = 6

197 / 10 = 19             197 % 10 =  7
```

Processing also provides other numeric operators, including operations that can be applied to integer data at the individual bit level. In the following descriptions, b, b_1, and b_2 denote binary digits (0 or 1); x and y denote integers.

Table 3-4 Bitwise Operators

<table>
<tr><th>Operator</th><th>Operation</th><th>Description</th></tr>
<tr><td>~</td><td>bitwise negation</td><td>~b is 0 if b is 1; ~b is 1 if b is 0</td></tr>
<tr><td>&</td><td>bitwise and</td><td>b_1 & b_2 is 1 if both b_1 and b_2 are 1; it is 0 otherwise</td></tr>
<tr><td>|</td><td>bitwise or</td><td>b_1 | b_2 is 1 if either b_1 or b_2 or both are 1; it is 0 otherwise</td></tr>
<tr><td>^</td><td>bitwise exclusive or</td><td>b_1 ^ b_2 is 1 if exactly one of b_1 or b_2 is 1; it is 0 otherwise</td></tr>
<tr><td><<</td><td>bitshift left)</td><td>x << y is the value obtained by shifting the bits in x y positions to the left</td></tr>
<tr><td>>></td><td>bitshift right</td><td>x >> y is the value obtained by shifting the bits in x y positions to the right *</td></tr>
</table>

Note: There is also an *unsigned right shift operator* >>> that fills the vacated bit positions at the left with 0s. >> is a *signed* right-shift operator that fills these positions with the sign bit of the integer being shifted.

To illustrate the behavior of these operators, the statements[6]

```
byte i = 6;         // 00000110
println(i | 4);     // 00000110 OR  00000100 = 00000110
println(i & 4);     // 00000110 AND 00000100 = 00100
println(i ^ 4);     // 00000110 XOR 00000100 = 00010
println(i << 1);    // 00000110 LS  1        = 00001100
println(i >> 1);    // 00000110 RS  1        = 00000011
println(~i);        // NEG 00000110          = 11111001
```

produce this output:

```
6
4
2
12
3
-7
```

In practice, such operations are used by methods that must inspect memory or interact directly with a computer's hardware, such as low-level graphics methods or operating system methods.

Operator Precedence. The order in which operators in an expression are applied is determined by a characteristic known as **operator precedence** (or **priority**): *In an expression involving several operators the operators `*`, `/`, and `%` have higher precedence than (i.e., are applied before) the operators `+` and `-`.* Thus, in the expression

```
2 + 3 * 5
```

`*` has higher precedence than `+`, so the multiplication is performed before the addition; therefore, the value of the expression is 17.

Operator Associativity. In Processing the binary operators `+`, `-`, `*`, `/`, and `%` are all **left-associative** operators, which means that in an expression containing operators with the same priority, the left operator is applied first. Thus,

```
9 - 5 - 1
```

is evaluated as

```
(9 - 5) - 1 = 4 - 1 = 3
```

Associativity is also used in more complex expressions containing different operators of the same priority. For example, consider

[6] Recall from chapter 1 that the `println()` method prints the value its argument on the text output console.

```
7 * 10 - 5 % 3 * 4 + 9
```

There are three high-priority operations, `*`, `%`, and `*`, and so left associativity causes the leftmost multiplication to be performed first, giving the intermediate result

```
70 - 5 % 3 * 4 + 9
```

`%` is performed next, giving

```
70 - 2 * 4 + 9
```

and the second multiplication is performed last, yielding

```
70 - 8 + 9
```

The two remaining operations, `-` and `+`, are equal in priority, and so left associativity causes the subtraction to be performed first, giving

```
62 + 9
```

and then the addition is carried out, giving the final result

```
71
```

Unary Operators. The operators `+` and `-` can also be used as **unary operators** (i.e., they can be applied to a single operand); for example, `-x` and `+34` are allowed. Similarly, the expression `3 * -4` is a valid Processing expression, producing the value `-12`. Unary operations have higher priority than `*`, `/`, and `%`. Thus, the integer expression:

```
-6 * +2 / -3
```

produces the value `+4`.

Using Parentheses. Parentheses can be used to change the usual order of evaluation of an expression as determined by precedence and associativity. Parenthesized subexpressions are first evaluated in the standard manner, and the results are then combined to evaluate the complete expression. If the parentheses are "nested" — that is, if one set of parentheses is contained within another — the computations in the innermost parentheses are performed first.

To illustrate, consider the expression

```
(7 * (10 - 5) % 3) * 4 + 9
```

The subexpression `(10 - 5)` is evaluated first, producing

```
(7 * 5 % 3) * 4 + 9
```

Next, the subexpression `(7 * 5 % 3)` is evaluated left to right, giving

```
(35 % 3) * 4 + 9
```

followed by

```
2 * 4 + 9
```

Now the multiplication is performed, giving

```
8 + 9
```

and the addition produces the final result

```
17
```

Care should be taken in writing expressions containing two or more operations to ensure that they are evaluated in the order intended. Even though parentheses may not be required, they should be used freely to clarify the intended order of evaluation and to write complicated expressions in terms of simpler expressions. It is important, however, that the parentheses balance — *for each left parenthesis, a matching right parenthesis must appear later in the expression* — since an unpaired parenthesis will result in a compilation error.

Numeric Methods. In addition to literals, constants, and variables, an operand in an expression may also be a value computed by a method.[7] For example, we have used the `random()` method in some of our examples in earlier chapters to generate random numbers. Table 3-5 lists the more commonly used methods provided in Processing. They, along with others, are described in detail in Processing's web reference library (http://www.processing.org/reference). Unless noted otherwise, each of these methods takes arguments of type `float` and returns a value of type `float`.

Table 3-5 Some Processing Methods

Method	Description
`abs(v)`	Absolute value of *v* (integer or real)
`pow(x, y)`	x^y
`sqrt(x)`	Square root of *x*
`ceil(x)`	Smallest `float` ≥ *x* that is equal to an integer
`floor(x)`	Largest float ≤ *x* that is equal to an integer
`rint(x)`	`int` value closest to *x*
`round(x)`	`long` value closest to *x* (an `int` if *x* is `float`)
`max(v, w)`	Maximum of *v* and *w* (integer or real)
`min(v, w)`	Minimum of *v* and *w* (integer or real)
`exp(x)`	e^x
`log(x)`	Natural logarithm of *x*

[7] Methods that return values (in contrast to void methods that do not) are also commonly called *functions* because they are used like functions in mathematics.

Thus, to calculate the square root of 5, we can write

```
sqrt(5.0)
```

As a more complicated example, if we wish to calculate $\sqrt{b^2 - 4a}$, we could write

```
sqrt(pow(b, 2) - 4.0 * a * c)
```

Note that if the value of the expression

```
pow(b, 2) - 4.0 * a * c
```

is negative, then an error results because the square root of a negative number is not defined.

Processing also provides the methods for trigonometric functions described in Table 3-6. These methods are very important and useful tools in graphics programming.[8]

Table 3-6 Trigonometric Methods

Method	Description
`sin(x)`	Sine of *x* radians
`cos(x)`	Cosine of *x* radians
`tan(x)`	Tangent of *x* radians
`atan2(x)`	Angle between –π and π whose tangent is *x*
`degrees(x)`	The degree equivalent of *x* radians
`radians(x)`	The radian equivalent of *x* degrees

3.3.2. Promotion and Casting

Implicit Type Conversion — Promotion. The programs shown above each contained a statement like

```
int width = 300, height = 225;
```

and later

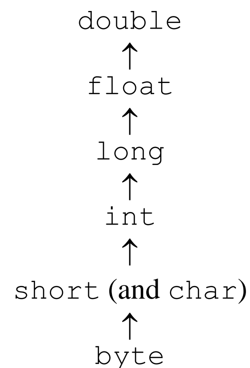
```
float aWidth = width/PHI, aHeight = height/PHI;
```

which were used to determine where we should draw lines to annotate a photo. In the last statement, integer values — `width` and `height` — were divided by a real value `PHI` and the resulting values were assigned to real variables — `aWidth` and `aHeight`. In such mixed-type expressions, Processing automatically converts an integer value to a real value as needed and performs the operation. Thus, when

[8] For a quick review of trigonometry, see the excellent tutorial at http://processing.org/learning/trig/ prepared by Professor Ira Greenberg.

the two division operations are to performed, they are first converted to 300.0/1.6180229887 and 225.0/1.6180229887 and the divisions carried out, producing real values for `aWidth` and `aHeight`. Note, however, that the values of `width` and `height` themselves do not change.

To understand how types are automatically converted to other types in expressions, suppose we arrange Processing's numeric data types in a list in descending order according to how much memory is used to store values of these types (see Table 3.1):



In mixed-type expressions, values whose type is lower in this listing will be converted to those that are higher before the computation is performed. Thus, for example, the result produced by `3/4` is 0, but `3.0/4` or `3/4.0` are computed as `3.0/4.0` which produces 0.75.

As a result of such type conversions referred to as *promotion*, `byte`, `short`, `int`, and `long` (and `char`) integer values and `float` and `double` real values can be freely intermixed in most numeric expressions, which is a great convenience for the programmer. However, promotion does have its limits. It is a *one-way* relationship as the arrows in the preceding diagram suggest. This means that we can write

```
float product = 1;
```

and the `int` value `1` will be converted to a `float` value and associated with `product`. But Processing will not compile either of the following statements and will output error messages like those shown:

```
int count = 1.0;  // ERROR: cannot convert float to int
int intVal = 1L;  // ERROR: cannot convert from long to int
```

If the type of one expression is the same as or can be promoted to the type of another expression, then the first expression is said to be *type-compatible* (or simply *compatible*) with the second expression. That is, the "can be promoted to" arrows in the preceding diagram also denote the “is compatible with” relationship. The one-way property of the arrows means that although an `int` is compatible with a `double`, a `double` is not compatible with an `int`. Also, the `boolean` type is not shown in the diagram because it is not compatible with any of the other primitive types.

Explicit Type Conversion — Casting. Sometimes it is necessary to convert an expression from one type to another. For such situations, Processing does provide a way that a programmer can do this. To illustrate, suppose we want to round a `double` value `doubleVal` to an `int`. One might think we need only use the `round()` method,

```
int intVal = round(doubleVal);  // Error!
```

but this statement does not compile because `round()` requires an argument of type `float`.

However, there is an alternative: use a **type-cast**, or simply, **cast**, where an expression of the form

```
typename(expression)
```

or

```
(typename)(expression)
```

can be used to convert *`expression`* to a value of type *`typename`*.[9] Using `int` for the typename will convert a real value to an integer by truncating the fractional part. For example, the `double` variable `doubleVal` is assigned the value 15.678,

```
double doubleVal = 15.678;
```

the statement

```
int intValue = int(doubleVal + 0.5);
```

or

```
int intValue = (int)( doubleVal + 0.5);
```

will add 0.5 to `doubleVal` giving 16.178 after which the type-cast will truncate the fractional part (.178) and assign the value 16 to `intValue`. Similarly, if `doubleVal` had the value 15.378,

```
double doubleVal = 15.378;
```

adding 0.5 produces 15.878, so the value 15 would be assigned to `intValue`.

3.3.3. Assignment Expressions

In several of our examples we have used an *assignment statement* of the form

```
variable = expression;
```

which uses the assignment operator (=) to change the value of a variable. In fact, what we were actually doing was converting an **assignment expression** of the form

```
variable = expression
```

to a statement by appending a semicolon.[10] In this assignment expression, *`expression`* must be type-compatible with *`variable`*. When it is executed,

---

[9] The second form for type-casting is the one used in Java.

1. *`expression`* is evaluated to produce some value *`v`*;
2. the value of *`variable`* is changed to *v*, and
3. *`v`* is the value of the entire expression.

For example, if `xValue` and `yValue` are declared by

```
float xValue = 0, yValue = 0;
```

then memory locations are allocated to `xValue` and `yValue` and they are each defined with the value zero. We might picture this as follows:

xValue	0.0
yValue	0.0

Now consider the assignment statements:

```
xValue = 25.0;
yValue = Math.sqrt(xValue);
```

The first statement changes the value of `xValue` to 25.0,

xValue	25.0
yValue	0.0

and then (using the value of `xValue`), the second statement changes the value of `yValue` to 5.0:

xValue	25.0
yValue	5.0

It is important to remember that for *primitive* types, an assignment statement is a *replacement* statement. Some beginning programmers forget this and write an assignment statement like

```
a = b;
```

when the statement

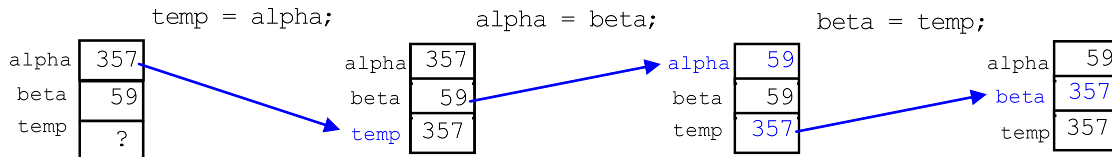
```
b = a;
```

is intended. These two statements produce very different results: The first assigns the value of `b` to `a`, leaving `b` unchanged, and the second assigns the value of `a` to `b`, leaving `a` unchanged.

[10] In fact, we could do this with any expression. For example, `1 + 2;` and `-3.1416;` are valid statements; they just doesn't do anything very interesting!

To illustrate further the replacement property of an assignment, suppose that the integer variables `alpha` and `beta` have values `357` and `59`, respectively, and that we wish to interchange these values. For this, we use a third integer variable `temp` to store the value of `alpha` while we assign `beta`'s value to `alpha`; then we can assign this stored value to `beta`.

```
temp = alpha;
alpha = beta;
beta = temp;
```



In a valid assignment statement, the variable whose value is to be changed must appear to the left of the assignment operator (=), and a type-compatible expression must appear on the right. The following table shows several invalid Processing assignments along with explanations of why they are not valid. The variables `x`, `y`, `number`, and `grade` have the following declarations:

```
float x, y;
int number;
char grade;
```

Table 3-7 Invalid Processing assignments

Statement	Error
`5 = number;`	The left operand of an assignment must be a variable.
`x + 3.5 = y;`	The left operand of an assignment cannot be an expression.
`grade = "ABCC";`	Type `String` is not compatible with type `char`.
`number = x;`	Type `double` is not compatible with type `int`.
`number = false;`	Type `boolean` is not compatible with any other type.

Assignment as an Operation. We have seen that an assignment

`variable = expression`

produces three actions:

1. *`expression`* is evaluated to produce some value *v*;
2. the value of *`variable`* is changed to *v*, and
3. *v* is the value of the entire assignment expression.

Thus far in our discussion, we have concentrated on actions (1) and (2), but we now turn our attention to action (3) in this description.

In the expression

```
2 + 3
```

+ is the operator, its operands are `2` and `3`, and it produces the value `5`. Similarly, in the assignment

```
number = 4
```

= is the operator, its operands are `num` and `4`, and it produces the value 4; as a "side effect" it also stores the value 4 in `number`'s memory location. As another example, suppose the value of `number` is `4` and consider the expression

```
number = number * 2
```

The `*` is applied first because it has higher precedence (the priority of the = operator is lower than almost all other operators),

```
number = (number * 2)
```

producing the result `8`. The assignment thus becomes

```
number = 8
```

which changes the value of `number` to 8 and produces the result 8. Although we are usually most interested in the side effect of the assignment — of changing the value of `number` to 8 — *it is important to remember that the assignment operator = is a value-producing binary operator whose result is the value assigned to the left operand*

Chaining Assignment Operators. Because = is a value-producing operator, several assignment operators can be chained together in a single statement such as

```
x = y = 2.5;
```

which is equivalent to the two separate statements

```
y = 2.5;
x = y;
```

Unlike most of the arithmetic operators, the assignment operator = is **right-associative**, so that in the statement

```
x = y = 2.5;
```

the rightmost = is applied first,

```
x = (y = 2.5);
```

which changes the value of `y` to 2.5; and produces the value assigned to `y` (i.e., 2.5). The assignment thus becomes

```
x = (2.5);
```

which changes the value of `x` to 2.5. It also produces the assigned value 2.5 as its result so the statement becomes

```
2.5;
```

which is a valid Processing statement; it just doesn't do anything.

Because of right-associativity and the value-producing property of =, chained assignments can be used to assign the same value to a group of variables; for example,

```
a = b = c = d = 1;
```

will set each of `d`, `c`, `b`, and `a` to 1.

Increment and Decrement Operations. Algorithms often contain instructions of the form

"Increment *counter* by 1."

One way to encode this instruction in Processing is

```
counter = counter + 1;
```

Such a statement in which the same variable appears on both sides of the assignment operator often confuses beginning programmers. Although we read English sentences from left to right, execution of this statement goes from right to left because + has higher priority than =:

1. `counter + 1` is evaluated
2. This value is assigned to `counter` (overwriting its previous value).

For example, if `counter` has the value `16`, then

1. The value of `counter + 1` (`16 + 1` = `17`), is computed; and
2. This value is assigned as the new value for `counter`:

As we have seen, the old value of the variable is lost because it is replaced with a new value.

This kind of assignment (i.e., incrementing a variable) occurs so often that Processing provides a special unary **increment operator ++** for this operation. It can be used as a postfix operator,

`variableName++`

or as a prefix operator,

`++variableName`

where `variableName` is an integer variable whose value is to be incremented by 1. Thus, the assignment statement

```
counter = counter + 1;
```

can also be written

```
counter++;
```

or

```
++counter;
```

The difference between the postfix and prefix use of the operator is subtle. If the increment operator is being used simply to increment a variable as a stand-alone statement, then it does not matter whether the prefix or postfix form is used. Both of these statements produce exactly the same result; namely, the value of `counter` is incremented by 1.

To explain the difference, consider the following program segments where `counter`, `number1`, and `number2` are `int` variables:

Code	Output
`//POSTFIX:  Use first, then increment`	
`counter = 10;`	
`println("counter = " + counter);`	`counter = 10`
`number1 = counter++;`	
`println("number1 = " + number1);`	`number1 = 10`
`println("counter = " + counter);`	`counter = 11`

and

Code	Output
`//PREFIX:  Increment first, then use`	
`counter = 10;`	
`println("counter = " + counter);`	`counter = 10`
`number1 = ++counter;`	
`println("number1 = " + number1);`	`number1 = 11`
`println("counter = " + counter);`	`counter = 11`

Note that after execution of both sets of statements, the value of `counter` is 11. However, in the first set of assignments, the value assigned to `number1` is 10, whereas in the second set of assignments, the value assigned to `number1` is 11. To understand this difference, we must remember that increment expressions are "shortcut" assignment expressions and thus produce values. If `counter` has the value 10, then in the *prefix* expression

```
++counter
```

`counter` is incremented (to 11) and *the value produced by the expression is the incremented value* (11). By contrast, if `counter` again has the value `10`, then in the *postfix* expression

```
counter++
```

`counter` is still incremented (to 11), but *the value produced by the expression is the original value* (10). Put differently, the assignment

```
number2 = ++counter;
```

is equivalent to

```
counter = counter + 1;
number2 = counter;
```

while the assignment

```
number1 = counter++;
```

is equivalent to

```
number1 = counter;
counter = counter + 1;
```

It does not matter whether the prefix or postfix form is used if the increment operator is being used simply to increment a variable as a stand-alone statement:

```
counter++;
```

or

```
++counter;
```

Both of these statements have the same side-effect; namely, the value of `counter` is incremented by 1.

Just as you can increment a variable's value with the `++` operator, you can decrement the value of a variable (i.e., subtract 1 from it) using the **decrement operator** (**--**), For example, the assignment statement

```
counter = counter - 1;
```

can be written more compactly as

```
counter--;
```

(or `--counter;`). The prefix and postfix versions of the decrement operator behave in a manner similar to the prefix and postfix versions of the increment operator.

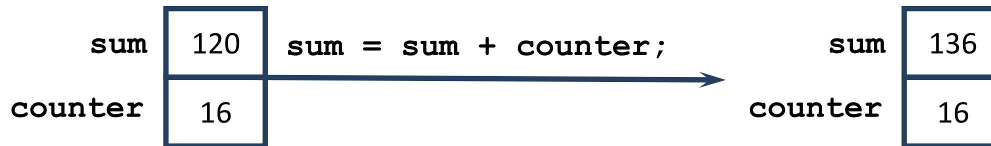
Other Assignment Shortcuts. The increment and decrement operations are special cases of a more general assignment that changes the value of a variable using some expression that involves its original value. For example, the instruction

"Add *counter* to *sum*"

implicitly changes the value of *sum* to the value of *sum* + *counter*. This can be encoded in Processing as

```
sum = sum + counter;
```

The following diagram illustrates this for the case in which the integer variables `sum` and `counter` have the values `120` and `16`, respectively.



This operation occurs so frequently that Processing provides special operators for it. Instead of writing

```
sum = sum + counter;
```

we can write

```
sum += counter;
```

to accomplish the same thing.

A similar shortcut is provided for each of the other arithmetic operators. For example,

```
number = number / 2;
```

can be written

```
number /= 2;
```

In general, a statement of the form

```
alpha = alpha Δ beta;
```

can be written :

```
alpha Δ= beta;
```

where Δ is any of the arithmetic operators `+`, `–`, `*`, `/`, or `%`, or one of the bitwise operators `&`, `|`, `^`, `<<`, `>>`, or `>>>`. Each of the following is, therefore, an acceptable variation of the assignment operator:

```
+=,  -=,  *=,  /=,  %=,  &=,  |=,  ^=,  <<=,  >>=,  >>>=
```

Like the regular assignment operator, each of these is right-associative and produces the value assigned as its result, which means that they can be chained together. This is not good programming practice, however, because it produces expressions for which it can be difficult to follow how they are evaluated. *Chaining such operators together should normally be avoided so that the readability of the program does not suffer. Programs that are cleverly written but difficult to read are of little use because they are too costly to maintain.*

3.3.4. Logical Expressions

In Processing, the primitive type `boolean` has two literals: `false` and `true`. A boolean expression is thus a sequence of operands and operators that combine to produce one of the boolean values, `false` or `true`.

The operators that are used in the simplest boolean expressions test some *relationship* between their operands. For example, the boolean expression

```
radius >= 0
```

which compares the (variable) operand `radius` and the (literal) operand `0` using the greater-than-or-equal-to relationship, produces the value `true` if the value of `radius` is nonnegative, but produces the value `false` if the value of `radius` is negative. Similarly, the Processing boolean expression

```
count == 5
```

tests the equality relationship between the operands `count` and `5`, producing the value `true` if the value of `count` is 5 and the value `false` otherwise.

> *Note: Be sure to use the == operator for equality comparisons, and not = (assignment). Trying to compare two values using the = operator will produce a compiler error.*

Operators like >= and == that test a relationship between two operands are called **relational operators**, and they are used in boolean expressions of the form

expression$_1$ *RelationalOperator* *expression*$_2$

where *expression*$_1$ and *expression*$_2$ are two compatible expressions, and the *RelationalOperator* may be any of the following:

Relational Operator	Relation Tested
`<`	Is less than
`>`	Is greater than
`==`	Is equal to
`!=`	Is not equal to
`<=`	Is less than or equal to
`>=`	Is greater than or equal to

These relational operators can be applied to operands of any of the primitive types: `char`, `int`, `float`, `double`, and so on. For example, if `x`, `a`, `b`, and `c` are of type `float`, and `numerator` and

`demoninator` are of type `int`, then the following are valid boolean expressions formed using these relational operators:

```
x < 5.2
b * b >= 4.0 * a * c
numerator == 50
denominator != 0
```

Compound `boolean` Expressions. Many relationships are too complicated to be expressed using only the relational operators. For example, a typical test score is governed by the mathematical relationship

$$0 \leq \text{test score} \leq 100$$

which is true if the test score is between 0 and 100 (inclusive), and is false otherwise. However, this relationship *cannot* be correctly represented in Processing by the expression

```
0 <= testScore <= 100
```

The reason is that these relational operators are left-associative, and so the preceding expression is processed by the Processing compiler as

```
(0 <= testScore) <= 100
```

The Processing compiler determines that the sub-expression

```
(0 <= testScore)
```

produces a `boolean` value, which it then tries to use as an operand for the second <= operator, giving the expression

```
(aBooleanValue <= 100)
```

At this point, the compiler generates an error, because `boolean` and `int` values are not compatible and thus cannot be compared (even with a cast).

To avoid this difficulty, we must rewrite the mathematical expression

$$0 \leq \text{test score} \leq 100$$

in a different form:

$$(0 \leq \text{test score}) \text{ and } (\text{test score} \leq 100)$$

This expression can be correctly coded in Processing, because Processing provides **logical operators** that combine boolean expressions to form **compound boolean expressions**. These operators are defined as follows:

Logical Operator	Logical Expression	Name of Operation	Description
!	`!p`	*Not (Negation)*	`!p` is false if `p` is true; `!p` is true if `p` is false.
&&	`p && q`	*And (Conjunction)*	`p && q` is true if both `p` and `q` are true; it is false otherwise.
||	`p || q`	*Or (Disjunction)*	`p || q` is true if either `p` or `q` or both are true; it is false otherwise.

These definitions are summarized by the following **truth tables,** which display all possible values for two conditions `p` and `q` and the corresponding values of the logical expression:

p	!p
true	false
false	true

p	q	p && q	p || q
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

We can thus use the `&&` operator to represent the mathematical expression

$$(0 \leq \text{test score}) \text{ and } (\text{test score} \leq 100)$$

by the compound boolean expression

```
(0 <= testScore) && (testScore <= 100)
```

This expression will correctly evaluate the relationship between `0`, `testScore`, and `100`, for all possible values of `testScore`.

Short-Circuit Evaluation. An important feature of the `&&` and `||` operators is that they do not always evaluate their second operand. For example, if `p` is false, then the condition

```
p && q
```

is `false`, regardless of the value of `q`, and so Processing does not evaluate `q`. Similarly, if `p` is true, then the condition

```
p || q
```

is `true`, regardless of the value of `q`, and so Processing does not evaluate `q`. This approach is called **short-circuit evaluation**, and has two important benefits:

1. One `boolean` expression can be used to *guard* a potentially unsafe operation in a second `boolean` expression
2. A considerable amount of time can be saved in the evaluation of complex conditions

As an illustration of the first benefit, consider the boolean expression

```
(n != 0) && (x < 1.0 / n)
```

No division-by-zero error can occur in this expression, because if `n` is 0, then the first expression

```
(n != 0)
```

is `false` and the second expression

```
(x < 1.0 / n)
```

is not evaluated. Similarly, no division-by-zero error will occur in evaluating the condition

```
(n == 0) || (x >= 1.0 / n)
```

because if `n` is 0, the first expression

```
(n == 0)
```

is `true` and the second expression is not evaluated.

Operator Precedence. A `boolean` expression that contains an assortment of arithmetic operators, relational operators, and logical operators is evaluated using the following precedence (or priority) and associativity rules:

Operator	Priority	Associativity
`()`	higher	Left
`!`, `++`, `--`, − (unary), + (unary) (*type-cast*), `new`		Right
`/`, `*`, `%`		Left
`+`, `-`		Left
`<`, `>`, `<=`, `>=`		Left
`==`, `!=`		Left
`&&`		Left
`||`		Left
`=`, `+=`, `*=`, . . .	lower	Right

An operator with a higher priority number has higher precedence and is applied before an operator with a lower priority number.

Because it is difficult to remember so many precedence levels, it is helpful to remember the following:

- Parenthesized subexpressions are evaluated first.
- `*`, `/`, and `%` have higher precedence than + and `-`.
- `!` is the highest-precedence logical operator.
- Every relational operator has higher precedence than the logical operators `&&` and `||`.
- Numeric operators have higher precedence than relational and/or logical operators (except `!`).
- Use parentheses for all the other operators to clarify the order in which they are applied.

To illustrate, consider the boolean expression:

```
x - (y + z) < a / b + c
```

where `x`, `y`, `z`, `a`, `b`, and `c` are all of type `double`. The parenthesized subexpression `(y + z)` is evaluated first, producing an intermediate real value *`v1`* and the expression becomes

```
x - v1 < a / b + c
```

Of the remaining operators, `/` has the highest priority, so it is applied next producing some intermediate value *`v2`*:

```
x - v1 < v2  + c
```

In the resulting expression, `-` and + are applied next, from left to right (because of left-associativity), producing some intermediate values *`v3`* and *`v4`*:

```
v3 < v4
```

Finally, the < operator is used to compare the last two (real) intermediate values and produces the value of the expression (`false` or `true`).

3.3.5. Character and String Expressions

As we have seen, Processing uses the type `char` to represent *individual characters*. This includes the uppercase letters `A` through `Z`; lowercase `a` through `z`; common punctuation symbols such as the semicolon (`;`), comma (`,`), and period (`.`); and special symbols such as `+`, =, and >.

We have also seen that `char` *literals* are usually written in Processing as single character symbols enclosed in apostrophes (or single quotes). For example,

```
'A', '@', '3', '+'
```

are all examples of Processing `char` literals. The apostrophe is thus used to *delimit* (i.e., surround and distinguish) `char` literals from other items in a Processing program.

Using an apostrophe as a delimiter raises the question, *What is the character literal for an apostrophe*? A similar question arises for characters such as the newline character, for which there is no corresponding symbol; newline characters "move" the output from the end of one line to the beginning of the next. For such characters that have a special purpose and cannot be described using the normal approach, Processing provides **escape sequences**, comprised of a backslash and another character. For example, the character literal for an apostrophe can be written as

```
'\''
```

and the newline character by

```
'\n'
```

Table 3-8 lists the escape sequences provided in Processing.

Table 3-8 Processing Character Escape Sequences

Character	Escape Sequence
Backspace (BS)	`\b`
Horizontal tab (HT)	`\t`
Newline or Linefeed (NL or LF)	`\n`
Carriage return (CR)	`\r`
Double quote (")	`\"`
Single quote (')	`\'`
Backslash (\\)	`\\`
With Unicode hexadecimal code *hhhh*	`\u`*hhhh*

As the last escape sequence indicates, Processing represents characters using the **Unicode** encoding scheme (see below) and any Unicode character can be generated from its hexadecimal code.

`char` Expressions. Compared to the other primitive types, there are very few predefined operations for objects of type `char`. Such objects can be defined and initialized in a manner similar to `int` and `double` objects; for example,

```
final char MIDDLE_INITIAL = 'C';
char       direction = 'N';          // N, S, E or W
```

Character values can be assigned in the usual manner,

```
direction = 'E';
```

but there are no shortcuts for `char` assignments comparable to those for numbers.

Values of type `char` can be compared using the relational operators. Such comparisons are performed using the Unicode numeric codes, so the expression

```
'A' < 'B'
```

produces `true` because the Unicode value for `A` (65) is less than the Unicode value for `B` (66). Similarly, the expression

```
'a' < 'b'
```

produces the value `true`, because the Unicode value for `a` (97) is less than the Unicode value for `b` (98).

The boolean expression

```
'a' < 'A'
```

is `false`, because the Unicode value for `a` (97) is not less than the Unicode value for `A` (65).

Compound boolean expressions can also be used to compare non-numeric values. For example, suppose we are solving a problem whose solution requires that a character variable `letter` must have an uppercase value. Such a condition might be expressed in Processing using a compound boolean expression:

```
('A' <= letter) && (letter <= 'Z')
```

The String Reference Type. The `String` type plays a very important role in Processing, in that most input and output is (by default) accomplished through `String` values. As a result, it is important to become familiar with Processing's `String` class and its capabilities. We introduce the `String` type here, and will study it in more detail in Chapter 6.

`String` Literals. We have seen that the `String` reference type allows a sequence of characters to be represented as a single object. As such, `String` literals consist of zero or more characters (including `char` escape sequences) surrounded by double-quotes. The following are thus valid `String` literals:

```
""
"123"
"\n\tEnter the rectangle's width and height: "
"\u0048\u0069\u0021"
```

As we noted earlier, most reference type values must be constructed using the `new` operator and a *constructor* method.The `String` type is unusual in this regard because Processing provides `String` literals that can be used to initialize or assign values to `String` variables. Thus instead of having to write declarations such as

```
String firstName = new String("Jane"),
```

we can simply use

```
String lastName = "Doe";
```

`String` Operations. Processing's `String` class provides a rich set of operations for manipulating `String` values. Perhaps the most frequently used is the `String` **concatenation (+) operator** which, given two `String` operands, produces a `String` consisting of the left operand followed by the right operand. For example, given the preceding declarations, we could use the statement

```
println(firstName + " " + lastName);
```

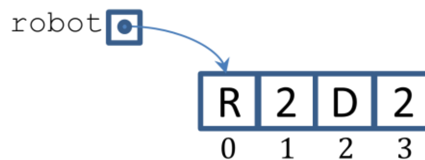
which first concatenates the `String` literals `"Jane"` and `" "` to produce `"Jane "` which is then concatenated with `"Doe"` and the resulting literal `"Jane Doe"` output to the text output window.

If one of the operands to the + operator is a `String` and the other is one of the primitive types, then the non-`String` operand is automatically converted to a `String` and + is treated as the concatenation operator. Thus, the statements

```
int two = 2;
String robot = "R" + two + "D" + two;
```

build a `String` object whose value is `"R2D2"`.

We might picture the storage for this object as follows:



As this picture indicates, a `String` is an **indexed** variable, which means that the individual characters within the `String` can be accessed via an integer index (shown below the character entries). In Processing, `String`'s `charAt()` method is used for this. The first character in a `String` always has index 0, so the expression

```
robot.charAt(0)
```

produces the `char` value `'R'`, the expression

```
robot.charAt(1)
```

produces the `char` value `'2'`, and so on.

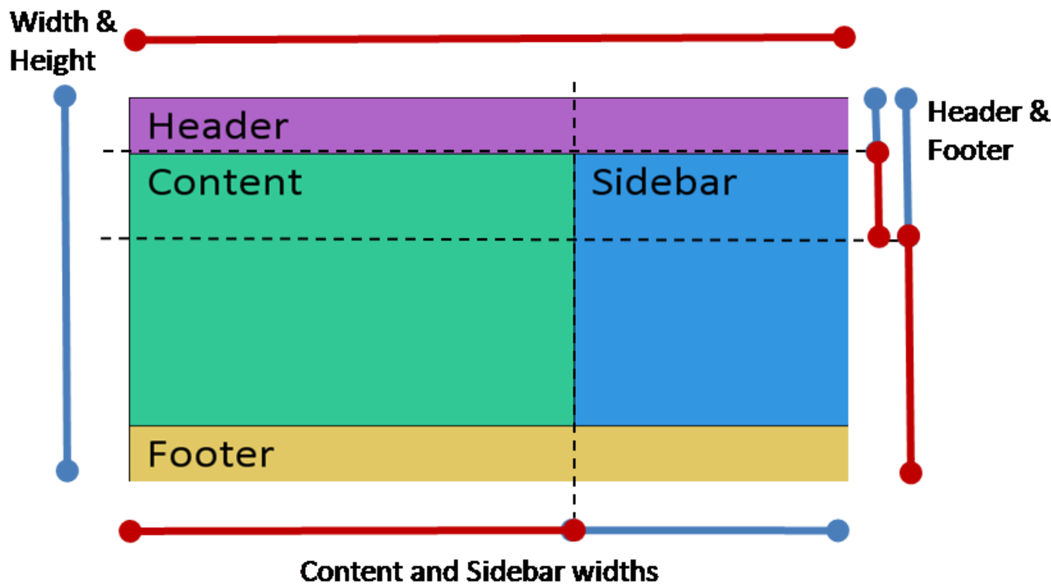
In addition to the concatenation operator and `charAt()` method, the Processing `String` class provides the `String`-related methods shown in Table 3-9 (See the `String` class API documentation for the complete list.)

Table 3-9 `String` Methods

Method	Description
`charAt()`	Returns the character at the specified index
`equals()`	Compares a string to a specified object
`indexOf()`	Returns the index value of the first occurrence of a character within the input string
`length()`	Returns the number of characters in the input string
`substring()`	Returns a new string that is part of the input string
`toLowerCase()`	Converts all the characters to lower case
`toUpperCase()`	Converts all the characters to upper case

3.4. Revisiting the Example

This section considers the use of the golden ratio in web design. The proportions that give pleasing photographic design can also give pleasing layouts for web content, so we'd like to build a web-page using the proportions we've used to annotate photographs. We'd like our web-page to include panes for a header, a footer, content and sidebar/navigation. A sketch of this goal, using the Phi-based ratio notation discussed in Section 3.1, is shown here.



This sketch shows that a number of the size relationships are proportional to the golden ratio, including the width and height of the web-page itself, the widths of the content and sidebar frames, and the height of the header and footer relative to the rest of the screen. The following algorithm lays out the design.

Given:

- The constant `PHI` represents the golden ratio (used as the ratio between the larger and smaller sections).

- The constant `PHI_FACTOR` represents 1 – 1/`PHI` (used as the ratio between the smaller and larger sections).
- The `width` and `height` of the display window are 600 and (600 / `PHI`).

Algorithm:

1. Create an display window that is `width` x `height` pixels.
2. Set `headHeight` = `height` * `PHI_FACTOR` * `PHI_FACTOR`.
3. Display a header rectangle `headHeight` high with the label "Header".
4. Display a labeled content rectangle with coordinates (0, `headHeight`), width `width/PHI` and height `height` – 2 * `headHeight`.
5. Display a labeled sidebar rectangle with coordinates (`width/PHI`, `headHeight`), width `width` * `PHI_FACTOR` and height `height` – 2*`PHI_FACTOR`.
6. Display a labeled footer rectangle with coordinates (0, `height` – `headHeight`) and with height `headHeight`.

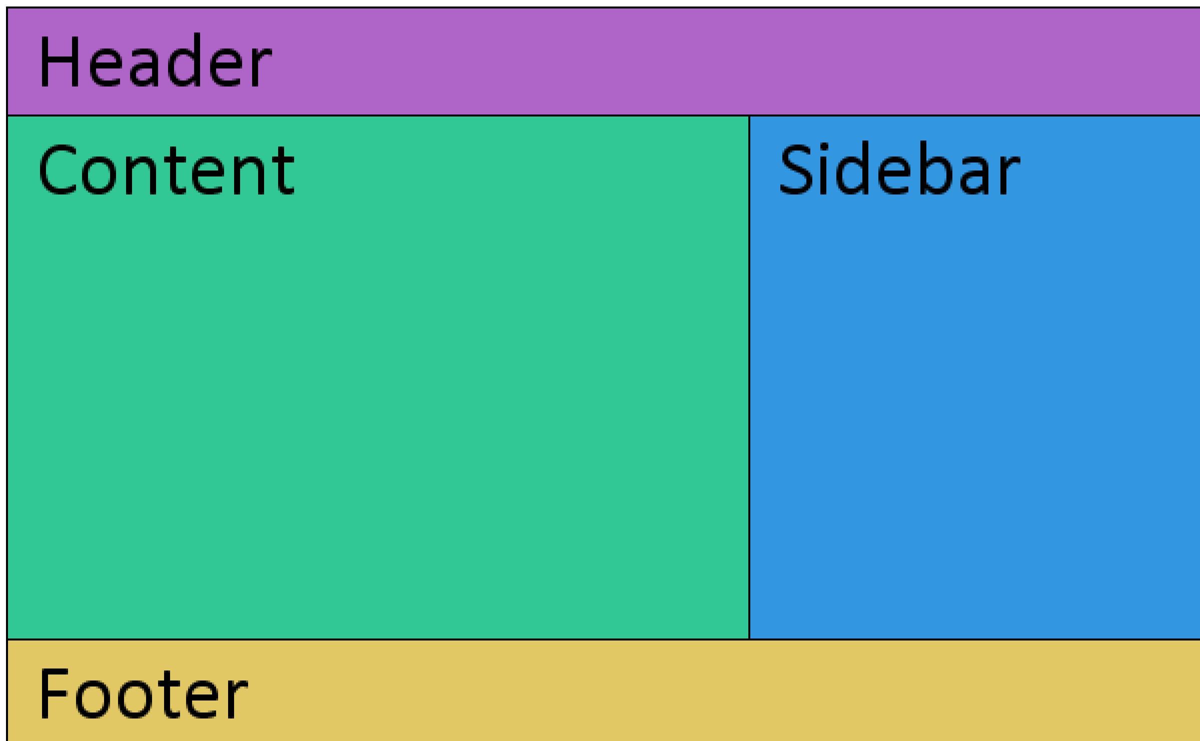
This algorithm can be implemented as follows:

```
/**
 * This sketch uses the golden ratio for web design.
 *
 * @author nyhl, jnyhoff, kvlinden, snelesen
 * @version Fall, 2011
 */
final float PHI = 1.6180339887, PHI_FACTOR = 1 - 1/PHI;
final int WIDTH = 600;

int height = int(WIDTH / PHI);
int fontSize = WIDTH/15, textPadding = WIDTH/40;
String caption1 = "Heading", caption2 = "Content",
       caption3 = "Sidebar", caption4 = "Footer";
 size(WIDTH, height);
 stroke(0);
 textFont(createFont("Calibri", fontSize));
 // Header
 fill(175, 100, 200);
 float headHeight = height * PHI_FACTOR * PHI_FACTOR;
 rect(0, 0, WIDTH, headHeight);
 fill(0);
 text(caption1, textPadding, fontSize);
 // Content
 fill(50, 200, 150);
 rect(0, headHeight, WIDTH/PHI, height - 2 * headHeight);
 fill(0);
 text(caption2, textPadding, headHeight + fontSize);
 // Sidebar
 fill(50, 150, 225);
 rect(WIDTH/PHI, headHeight,
      WIDTH*PHI_FACTOR, height-2*headHeight);
```

```
fill(0);
text(caption3, WIDTH/PHI + textPadding, headHeight + fontSize);

// Footer
fill(225, 200, 100);
rect(0, height - headHeight, WIDTH, headHeight);
fill(0);
text(caption4, textPadding, height - headHeight + fontSize);
```



The program uses several constants as we have used in the past. Additionally, the program uses several variables of type `int`: `fontSize` for the size of the font (Calibri) used for the captions; and `textPadding` for the amount of space to leave to the left of these captions. It also uses four `String` variables for the captions: `caption1`, `caption2`, `caption3`, and `caption4`.[11] Several of the operations described in this chapter are used to calculate the dimensions and locations of the rectangles and captions. Finally, the code uses the `createFont()` method to create a font object; this is a alternate method for creating fonts dynamically rather than using the font tool discussed in the previous chapter. This approach to building fonts has the advantage of allowing the program to determine the nature of the font, but the disadvantage of being much slower than using pre-built fonts.

---

[11] An argument could probably be made that all of the variables in this program should really be constants since they never change during the course of the execution of the program.